メッセージ通信
Message Oriented Communication

佐藤

分散システムの通信

- 通信レイヤとプロトコル
- Remote Procedure Call (RPC: 遠隔手続き呼び出し)
- Message-Oriented Middleware (MOM)
- data-streaming

通信の分類

- persistent communication
  - メッセージが通信中にどこかで保持されること
- transient communication
  - メッセージが通信中に消えてしまう可能性がある通信

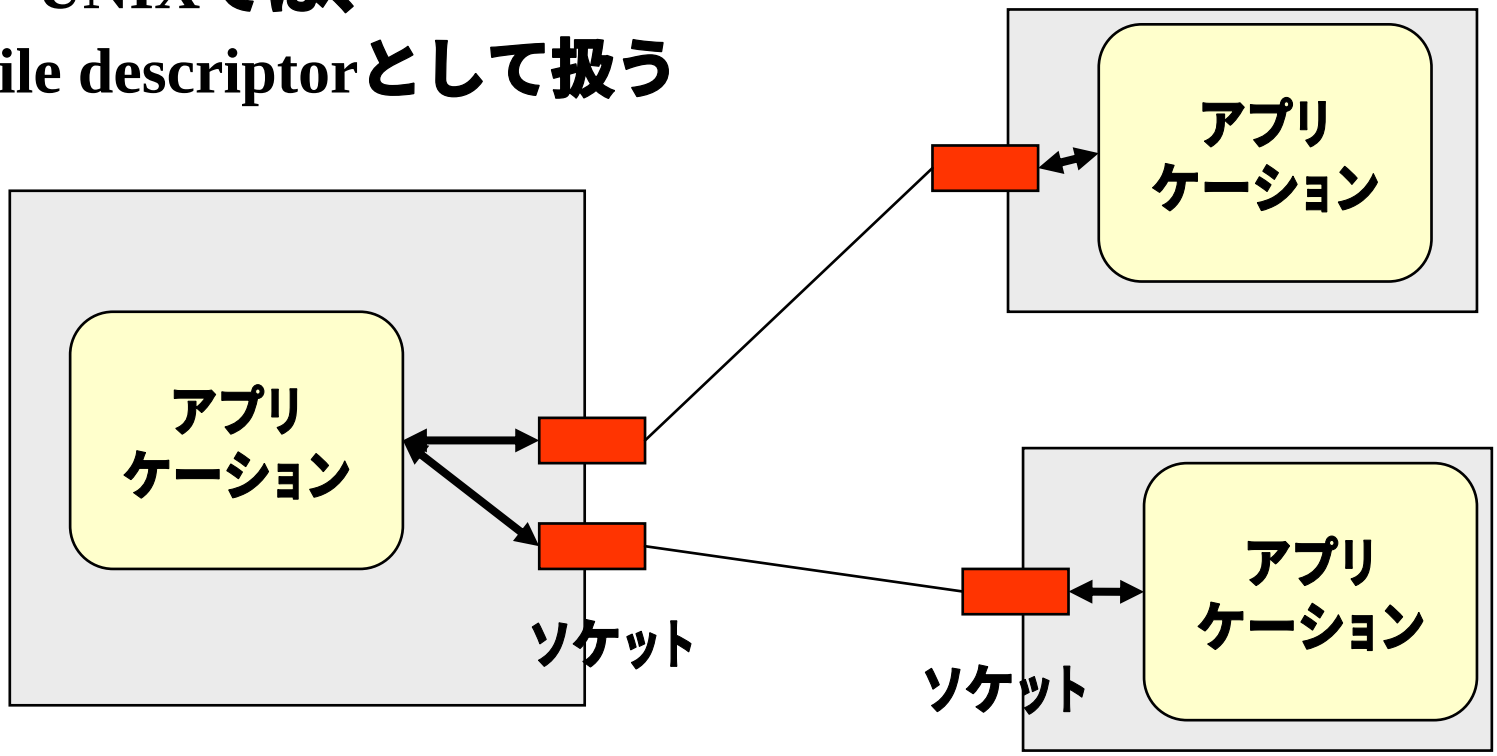
- asynchronous communication （非同期通信）
  - 送り手が、受け取りを待たずに、動作を始めること
- synchronous communication （同期通信）
  - 送り手が、相手が受け取るまでblockingして行う通信

Message-Oriented Transient Communication

- transport-layerでのMessage-Orientedな通信
  - Berkeley Sockets
  - MPI: Message Passing Interface

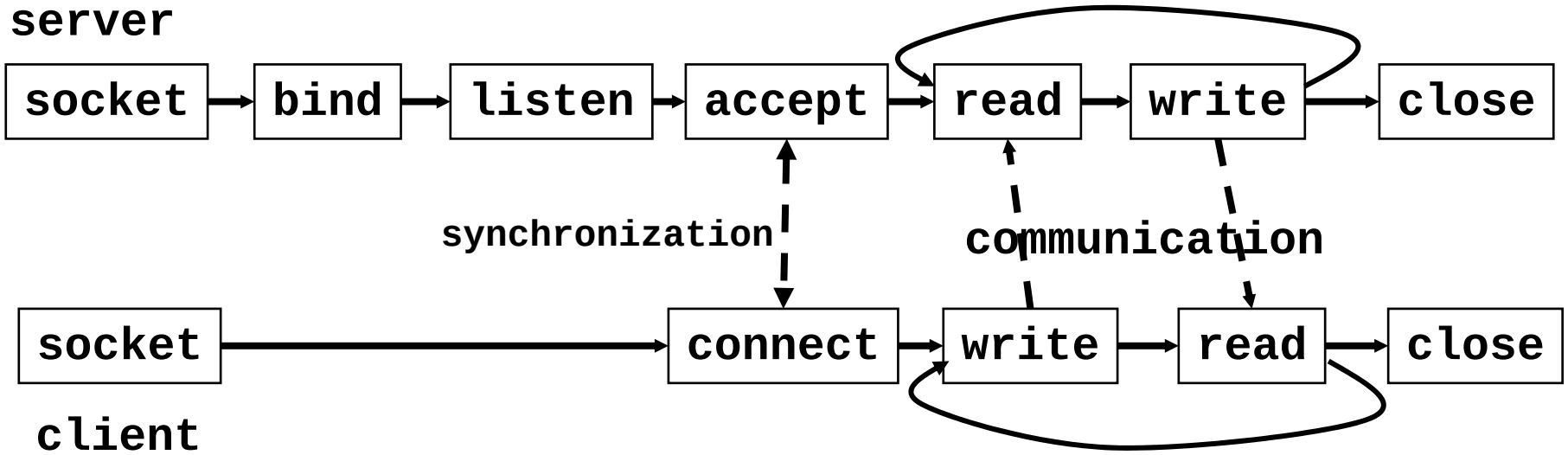
ソケット通信
（ネットワークプログラミング）

- socket　ソケット
  - TCPやUDPを行うためのend-point
  - UNIXでは、
    file descriptorとして扱う



ソケット・プログラミング

- 1970年代にBerkeley UNIXで導入されたのが最初。
- XTI, X/Open Transport Interface, (TLI: Transport Layer Interface): AT&Tで開発された。モデルは似ているが、いくつかのプリミティブで異なる



Socket Primitives for TCP/IP

- Socket : Create a new communication end point
- Bind: Attach a local address to a socket
- Listen: Announce willingnesss to accept connections
- Accept: Block caller until a connection request arrives
- Connect: Actively attempt to establish a connection
- Send: Send some data over the connection
- Receive: Receive some data over the connection
- Close: Release the connection

ネットワークプログラミング：ＴＣＰ

- サーバー側

```
s = socket(); /* socketを作る*/
bind(s,address); /* 名前を与える */
listen(s,backlog); /* backlogの指定 */
ss = accept(s); /* connectionが発生したら
          新しいfile descriptorを返す */
close(s); /* 必要なければ、もとのsはclose */
recv(ss,…); /* read 開始 */
```

- クライアント側

```
s = socket(); /* socketを作る*/
connect(s,address); /* connectionする*/
send(s,…); /* send開始 */
```

Server側(1)

```
// 省略
int my_fd;
struct sockaddr_in my_sin;
static int _setup_server_socket(struct sockaddr_in *sinp,
                                int port, int backlog);
int main(int argc,char *argv[])
{
    int sinlen;
    struct sockaddr_in client_sin;
    char buf[128];
    int r,s;
    int port;
    if(argc != 2){
        fprintf(stderr,"%s #port¥n",argv[0]);
        exit(1);
    }
    port = atoi(argv[1]);
    printf("server test program ... wait on port %d¥n",port);
   my_fd = _setup_server_socket(&my_sin,port,1);
    sinlen = sizeof(struct sockaddr_in);
    s = accept(my_fd,(struct sockaddr *)&client_sin,&sinlen);
    if(s < 0){
        perror("accept failed");
        exit(1);
    }
    while((r = read(s,buf,128)) >= 0){
        write(1,buf,r);
    }
    printf("terminated ...¥n");
    close(s);
    close(my_fd);
    exit(0);
```

Server側 (2)

```
static int _setup_server_socket(struct sockaddr_in *sinp,int port,
       int backlog)
{
    int sinlen,r;
    struct sockaddr_in sin;
    char hostname[MAXHOSTNAMELEN];
    struct hostent *hp;
    int fd;

    fd = socket(AF_INET, SOCK_STREAM, 0);
    if(fd < 0){
        perror("socket failed");
        exit(1);
    }
   r = gethostname(hostname,MAXHOSTNAMELEN);
    if(r < 0){
        perror("hostname");
        exit(1);
    }
    printf("hostname=%s¥n",hostname);

    hp = gethostbyname(hostname);
    if(hp == NULL){
        perror("gethostbyname");
        exit(1);
    }
     memset(&sin, 0, sizeof(sin));
    sin.sin_family = AF_INET;
    sin.sin_port = htons(port);
    bcopy(hp->h_addr,&sin.sin_addr.s_addr,hp->h_length);
```

Server側 (3)

```
    sinlen = sizeof(sin);

    r = bind(fd, (struct sockaddr *) & sin, sizeof(sin));
    if (r < 0){
        perror("bind");
        exit(1);
    }

    r = listen(fd,backlog);  /* set backlog */
    if (r < 0){
        perror("listen");
        exit(1);
    }

    r = getsockname(fd,(struct sockaddr *)sinp, &sinlen);
    if(r < 0){
        perror("getsockname");
        exit(1);
    }

    return fd;
}
```

Client側(1)

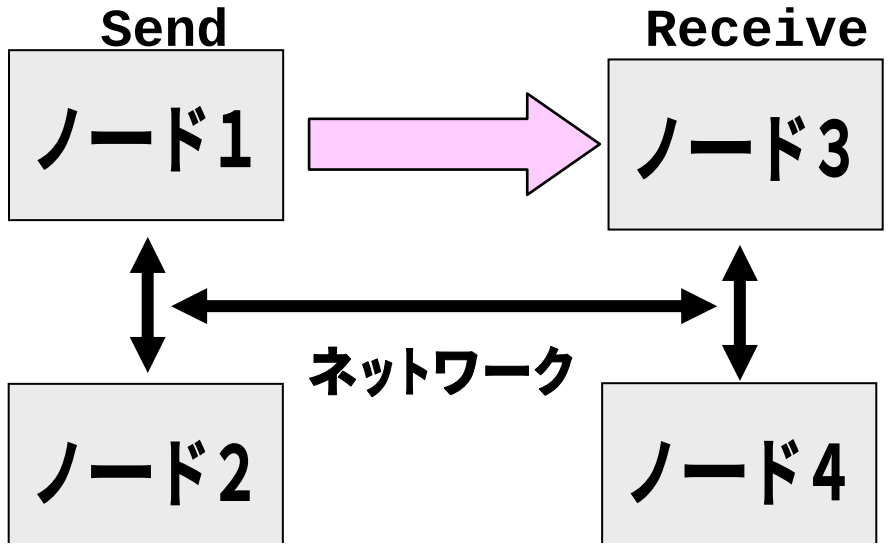
```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>

#ifndef MAXHOSTNAMELEN
#define MAXHOSTNAMELEN 256
#endif

int main(int argc,char *argv[])
{
    int r;
    struct sockaddr_in sin;
    char hostname[MAXHOSTNAMELEN];
    struct hostent *hp;
    int fd,port;
    char buf[128];
     if(argc != 3){
        fprintf(stderr,"%s: hostname port¥n");
        exit(1);
    }
    strcpy(hostname,argv[1]);
    port = atoi(argv[2]);
    printf("client test ... connect to %s:%d¥n",hostname,port);
    hp = gethostbyname(hostname);
    if(hp == NULL){
        perror("gethostbyname");
        exit(1);
    }
```

Client側(2)

```
    memset(&sin, 0, sizeof(sin));
     sin.sin_family = AF_INET;
     bcopy(hp->h_addr,&sin.sin_addr.s_addr,hp->h_length);
     sin.sin_port = port;

     fd = socket(AF_INET, SOCK_STREAM, 0);
     if(fd < 0){
         perror("socket failed");
         exit(1);
     }

     r = connect(fd,(struct sockaddr *)&sin,sizeof(sin));
     if(r < 0){
         perror("connect failed");
         exit(1);
     }

    sprintf(buf,"hello world...¥n");
     write(fd,buf,strlen(buf)+1);

     close(fd);
     exit(0);
}
```

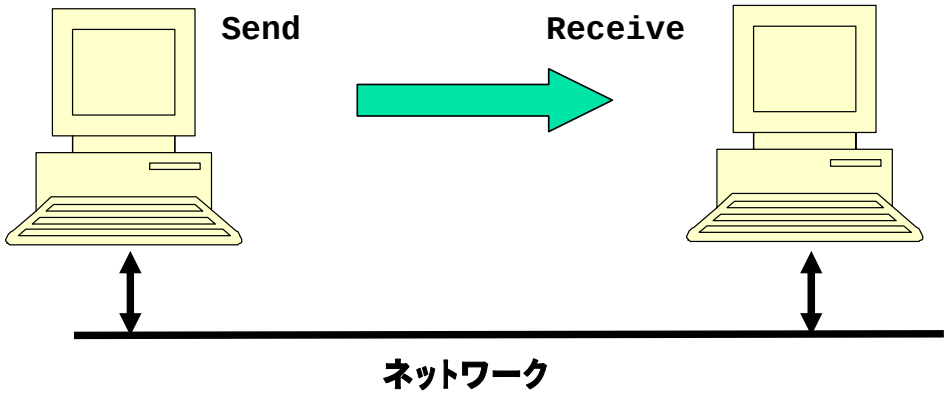
メッセージ通信プログラミング

- 分散メモリの一般的なプログラミングパラダイム
  - sendとreceiveでデータ交換をする
- 通信ライブラリ・レイヤ
  - POSIX　IPC, socket
  - TIPC (Transparent Interprocess Communication)
  - LINX (on Enea's OSE Operating System)
  - MCAPI (Multicore Communication API)
  - MPI (Message Passing Interface)



MPIによるプログラミング

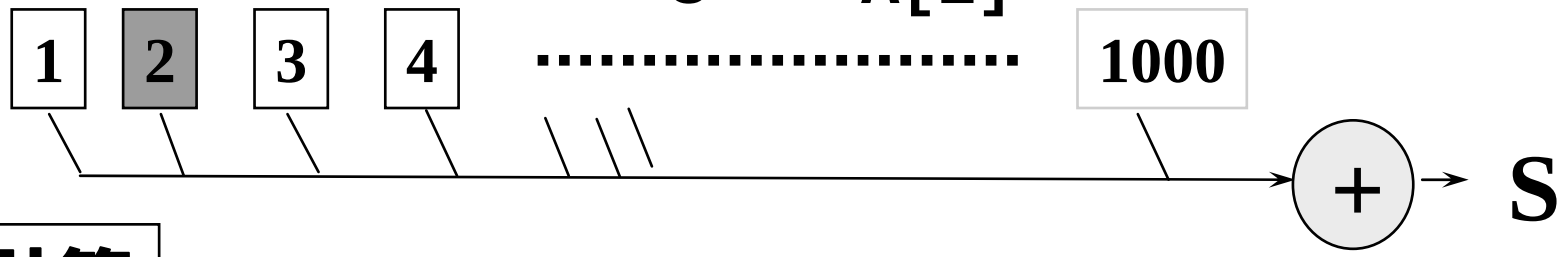
- MPI (Message Passing Interface)
  - おもに用途は、高性能科学技術計算
  - 分散メモリシステムにおける標準的なプログラミングライブラリ
- メッセージをやり取りして通信を行う
  - Send/Receive
  - ノード番号(rank)でやり取り
  - communicator
    - 通信しているノードの集合
- 集団通信もある
  - Reduce/Bcast
  - Gather/Scatter



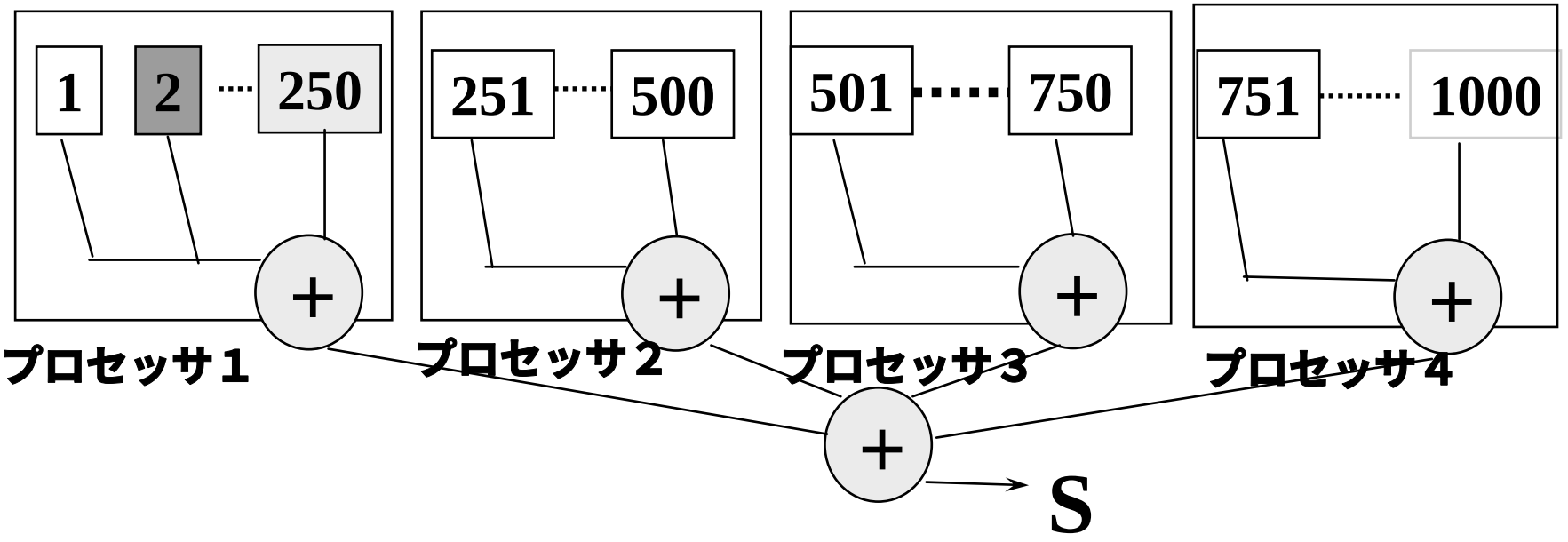
並列処理の簡単な例

逐次計算

```
for(i=0;i<1000; i++)
    S += A[i]
```



並列計算



メッセージ通信プログラミング

- 1000個のデータの加算の例

```
int a[250]; /* それぞれ、250個づづデータを持つ */

main(){     /* それぞれのプロセッサで実行される */
    int i,s,ss;
    s=0;
    for(i=0; i<250;i++) s+= a[i]; /*各プロセッサで計算*/
    if(myid == 0){    /* プロセッサ0の場合 */
        for(proc=1;proc<4; proc++){
            recv(&ss,proc);  /*各プロセッサからデータを受け取る*/
            s+=ss;    /*集計する*/
        }
    } else {   /* 0以外のプロセッサの場合 */
        send(s,0);  /* プロセッサ0にデータを送る */
    }
}
```

MPIでプログラミングしてみると

```
#include "mpi.h"
#include <stdio.h>
#define MY_TAG 100
double A[1000/N_PE];
int main( int argc, char *argv[])
{
    int n, myid, numprocs, i;
    double sum, x;
    int  namelen;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Get_processor_name(processor_name,&namelen);
    fprintf(stderr,"Process %d on %s¥n", myid, processor_name);

    ....
```

MPIでプログラミングしてみると

```
    sum = 0.0;
    for (i = 0; i < 1000/N_PE; i++){
       sum+ = A[i];
    }

    if(myid == 0){
       for(i = 1; i < numprocs; i++){
          MPI_Recv(&t,1,MPI_DOUBLE,i,MY_TAG,MPI_COMM_WORLD,&status)
            sum += t;
        }
    } else
            MPI_Send(&t,1,MPI_DOUBLE,0,MY_TAG,MPI_COMM_WORLD);
    /* MPI_Reduce(&sum, &sum, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_\
     MPI_Barrier(MPI_COMM_WORLD);
    ...
    MPI_Finalize();
    return 0;
}
```

Message-Oriented Persistent Communication

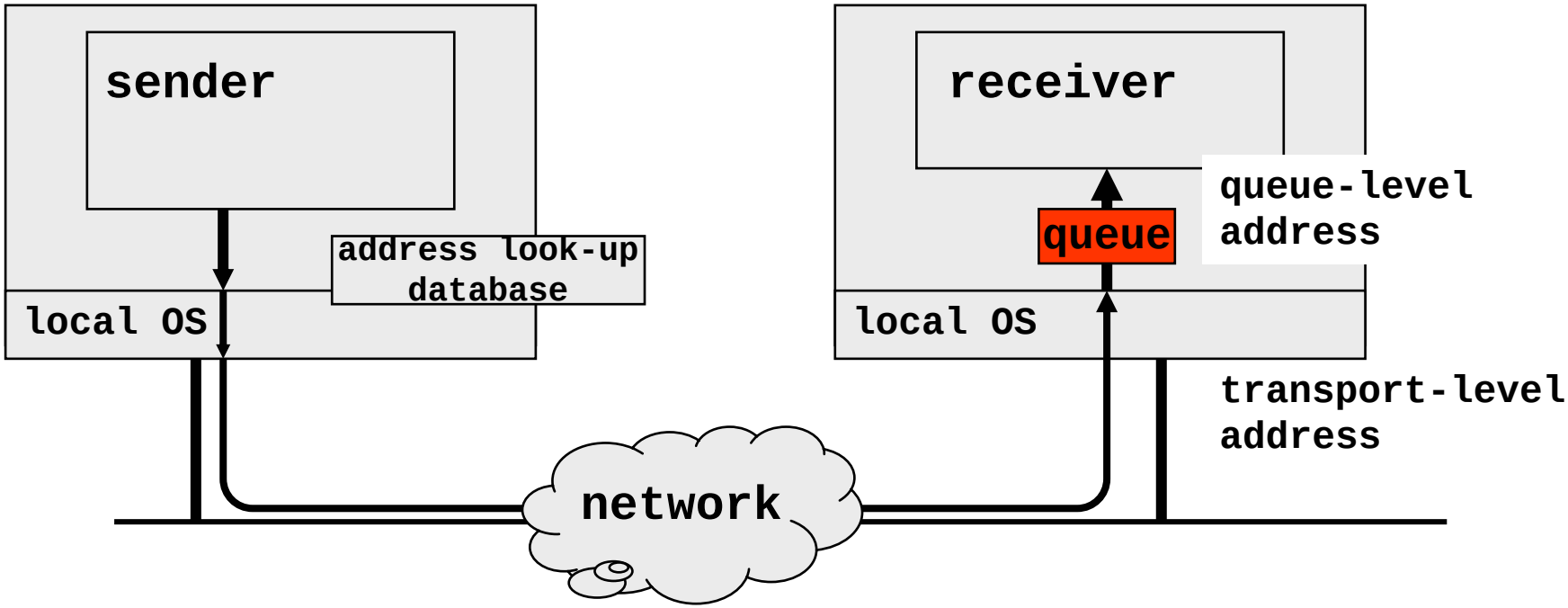
- **Message-queuing system, Message-Oriented Middleware (MOM)**
  - **Persistent and asynchronous**
  - **メッセージを保持しておくためのストレージがある**
  - **送り手、受け手が、activeでなくてもいい**
    - **loosely-coupled communication**
  - **ソケットとかMPIとの大きな違いは、実際の転送が数分単位でよい。（ソケットとかMPIだと、数秒単位）**
  - **典型的な例：メールシステム(SMTP)**

Primitive for Message queuing systems

- Put: Append a message to a specified queue
- Get: Block until the specified queue is non-empty, and remove the first message
- Poll: Check a specified queue for message, and remove the first. Never block.
- Notify: Install a handler to be called when a message is put into the specified queue.

General Architecture for Message queuing systems

- Mapping of queues to network location
  - transport level addressとqueue-level addressの変換が必要
  - DNS (Domain Name System/Server) in E-mail system
- Message Routerこの仕組みは、ルータが中間にはいってもよい
- Message Broker -- なんらかのメッセージを橋渡しをする仕組み



Stream-oriented communication

- Continuous Media (連続メディア)を扱う通信
  - データの時系列が中のデータを解釈するのに本質的なメディアのこと
  - 画像の通信(MPEG) -- 30 msec/imageの転送が必要
  - それに対する用語は、discrete media

- Data Stream
  - 基本は、TCP/IP
    - データの到着保障を行うプロトコル
  - asynchronous transmission mode (非同期通信モード)
    - データ間に時間的な制限がない転送モード
  - synchronous transmission mode (同期通信モード)
    - 最大の遅れ保障が定義されているような転送モード
  - isochronous transmission mode
    - 最大と最小の遅れ(bounded jitter)があるような通信モード
  - simple stream と complex stream
    - 画像と音声など

Streams and Quality of Service

- QoS (Quality of Service): 通信やサービスの品質
- どのように定義するか:
  - データ転送に必要なビットレート
  - セッションが設定されるまでの最大時間
    - つまり、アプリケーションがいつデータが送ることができるようになるか
  - end-to-endの遅延時間
    - データが受けて側で利用可能になるまで、どのくらいかかるか
  - 最大の遅延のばらつき
  - 最大のround-trip (往復）の遅延時間
- QoSの保障
  - bufferingをして、jitterによる影響を減らす

Stream Synchronization

- マルチメディアでは、complex streamにある複数のストリーム間の同期が重要になる
  - discrete media (JPEG?)とaudio data
  - MPEG (Motion Picture Expert Group) Streamのvideoとaudio
- MPEG2の例：
  - broadcast qualityのvideoを4から6Mbpsで転送できるように設計
  - 複数のcontinuous streamとdiscrete streamsのpacketを一つのstreamにマージ
  - それぞれのストリームは90KHzのシステムクロックのタイムスタンプがついたパケットになる
  - これらのストリームは、複数の可変長のパケットからなるprogram streamにmultiplex (多重化）される
  - 受け取る側はこれをdemultiplexして、復元

Multicast communication

- 同じデータを複数の相手に届ける通信
  - 似た言葉にbroadcastがある
  - インターネット放送などのアプリがあって、これから重要な通信機能
- いろいろなレベルのサポートが考えられる
  - これまで、多くはネットワークプロトコルの分野の課題であると考えられてきた
    - network-level or transport-level
    - IPレベルでのbroadcast
    - ここでの課題は、information dissemination (情報伝達）のpathをどのように設定するか
    - 実際には、ISPに膨大な管理コストがかかるので、余り積極的には行われていない。
  - アプリケーションレベルのMulticast

Application-level Multicast

- IPではなくて、アプリケーションレベルのネットワークすなわち、overlay network上でMulticastをサポートすること
  - overlay network : アプリケーションレベルのアドレス付けで作ったネットワーク
    - P2Pネットワーク
    - ファイル共有ソフトでも重要
  - overlayネットワーク上で、information disseminationのpathを決める
    - 例えば、spanning treeをつくるとか…
  - どのような経路で情報を伝達すると効率的か
    - epidemic protocols : ”流行”していくようなプロトコル
    - Gossip-based data dissemination
    - reputation-based …

まとめ

- **Message-Oriented Transient Communication**
  - **Berkeley Socket**
  - **MPI: Message Passing Interface**
- **Message-Oriented Persistent Communication**
  - **queuing system**
- **Stream-oriented communication**
  - **QoS**
  - **Stream Synchronization**
- **Multicast Communication**